

# Scalable Parallel Newton-Krylov Solvers for Discontinuous Galerkin Discretizations

Per-Olof Persson\*

*University of California, Berkeley, Berkeley, CA 94720-3840, U.S.A.*

We present techniques for implicit solution of discontinuous Galerkin discretizations of the Navier-Stokes equations on parallel computers. While a block-Jacobi method is simple and straight-forward to parallelize, its convergence properties are poor except for simple problems. Therefore, we consider Newton-GMRES methods preconditioned with block-incomplete LU factorizations, with optimized element orderings based on a minimum discarded fill (MDF) approach. We discuss the difficulties with the parallelization of these methods, but also show that with a simple domain decomposition approach, most of the advantages of the block-ILU over the block-Jacobi preconditioner are still retained. The convergence is further improved by incorporating the matrix connectivities into the mesh partitioning process, which aims at minimizing the errors introduced from separating the partitions. We demonstrate the performance of the schemes for realistic two- and three-dimensional flow problems.

## I. Introduction

Discontinuous Galerkin (DG) methods are capable of producing highly accurate numerical solutions to systems of conservation laws.<sup>1,2</sup> The low dispersion makes them attractive for problems with nonlinear interactions, wave propagation, or multiscale phenomena. Problems in aerodynamics are of particular interest, and much work has been done on the development of DG schemes and solvers for compressible flow.<sup>3-5</sup>

A fundamental drawback with these schemes is the high memory and computational cost, already at polynomial orders as low as 3 or 4. The total number of nodes is higher than a corresponding finite difference or continuous finite element method, but the main expense is due to the large number of connectivities generated between the unknowns. This results in expensive integrals to be computed for the discretization, and in very large systems of equations to be solved when using implicit methods. The high cost of DG methods has motivated several efforts into developing parallel implementations.<sup>6,7</sup>

Even though many parts of the computations in the DG method are local and hence natural for a parallel implementation, efficient implicit solvers require the use of preconditioners. Unfortunately, good preconditioners such as incomplete LU factorizations are non-local and of a sequential nature, which make them hard to parallelize. While a simple block-Jacobi approach scales better in parallel, its convergence properties are often insufficient.

In this paper, we investigate several strategies to develop effective parallel implementations. We consider iterative Krylov subspace methods with the block-ILU(0) preconditioner that we developed for serial implementations in Ref. 8, with optimal element orderings from the minimum discarded fill (MDF) algorithm. We show that the standard coloring technique used for parallelization of ILU essentially destroys the good properties of the MDF ordering, resulting in performance almost as poor as with the Jacobi preconditioner. Instead, we use a simple domain decomposition approach that is quite competitive for a wide range of problems. We also show that by partitioning the domain based on inter-element weights obtained from the discretized matrix, the performance loss from the ILU decomposition can be reduced.

---

\*Assistant Professor, Department of Mathematics and Lawrence Berkeley National Laboratory, University of California, Berkeley, Berkeley CA 94720-3840. E-mail: persson@berkeley.edu. AIAA Member. This work was supported by the Director, Office of Science, Computational and Technology Research, U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

## II. Problem Formulation

We consider a time-dependent system of conservation laws of the form,

$$\frac{\partial u}{\partial t} + \nabla \cdot F(u, \nabla u) = S(u, \nabla u) , \quad (1)$$

in a domain  $\Omega$ , with conserved state variables  $u$ , flux function  $F$ , and source term  $S$ . We eliminate the second order spatial derivatives of  $u$  by introducing additional variables  $q = \nabla u$ :

$$\frac{\partial u}{\partial t} + \nabla \cdot F(u, q) = S(u, q) , \quad (2)$$

$$q - \nabla u = 0 . \quad (3)$$

Next, we consider a triangulation  $\mathcal{T}_h$  of the spatial domain  $\Omega$  and introduce the finite element spaces  $V_h$  and  $\Sigma_h$  as

$$V_h = \{v \in [L^2(\Omega)]^m \mid v|_K \in [\mathcal{P}_p(K)]^m, \quad \forall K \in \mathcal{T}_h\} , \quad (4)$$

$$\Sigma_h = \{r \in [L^2(\Omega)]^{dm} \mid r|_K \in [\mathcal{P}_p(K)]^{dm}, \quad \forall K \in \mathcal{T}_h\} , \quad (5)$$

where  $\mathcal{P}_p(K)$  is the space of polynomial functions of degree at most  $p \geq 0$  on triangle  $K$ ,  $m$  is the dimension of  $u$  and  $d$  is the spatial dimension. We now consider DG formulations of the form: find  $u_h \in V_h$  and  $q_h \in \Sigma_h$  such that for all  $K \in \mathcal{T}_h$ , we have

$$\int_K q_h \cdot r \, dx = - \int_K u_h \nabla \cdot r \, dx + \int_{\partial K} \hat{u} r \cdot n \, ds , \quad \forall r \in [\mathcal{P}_p(K)]^{dm} , \quad (6)$$

$$\int_K \frac{\partial u_h}{\partial t} v \, dx - \int_K F(u_h, q_h) \cdot \nabla v \, dx = \int_K S(u_h, q_h) v \, dx - \int_{\partial K} \hat{F} \cdot n v \, ds , \quad \forall v \in [\mathcal{P}_p(K)]^m . \quad (7)$$

Here, the numerical fluxes  $\hat{F}$  and  $\hat{u}$  are approximations to  $F$  and  $u$ , respectively, on the boundary  $\partial K$  of the element  $K$ . We use Roe's scheme<sup>9</sup> for the inviscid fluxes and the compact discontinuous Galerkin CDG scheme<sup>10</sup> for the viscous fluxes.

It turns out that with the choices made for the numerical viscous fluxes, it is possible to eliminate the variables associated with  $q_h$  within each element and write the equations (6) and (7) as a system of coupled ordinary differential equations (ODEs) of the form:

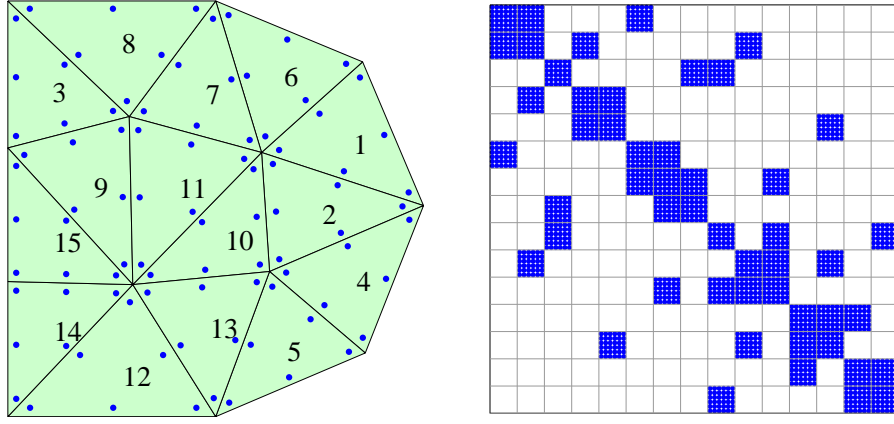
$$\mathbf{M} \dot{\mathbf{u}} = \mathbf{r}(\mathbf{u}) , \quad (8)$$

where  $\mathbf{u}(t)$  is a vector containing the degrees of freedom associated with  $u_h$ , which is represented using a nodal basis. The vector  $\dot{\mathbf{u}}$  denotes the time derivative of  $\mathbf{u}(t)$ ,  $\mathbf{M}$  is the mass matrix, and  $\mathbf{r}$  is the residual vector. We integrate (8) implicitly in time using the backward differentiation formulas (BDF)<sup>11</sup> or the diagonal implicit Runge-Kutta methods (DIRK).<sup>12</sup> Using Newton's method for solving the nonlinear systems of equations that arise, it is required to solve linear systems involving matrices of the form

$$\mathbf{A} \equiv \alpha_0 \mathbf{M} - \Delta t \frac{d\mathbf{r}}{d\mathbf{u}} \equiv \alpha_0 \mathbf{M} - \Delta t \mathbf{K} . \quad (9)$$

We assume that  $\alpha_0 = 1$ , which is the case for the first-order accurate backward Euler method, and approximately the case for higher-order methods using BDF and DIRK based schemes. In the next sections, we will study the solution of this system of equations using parallel iterative methods.

The system matrix  $\mathbf{A} = \mathbf{M} - \Delta t \mathbf{K}$  is sparse with a block-wise structure corresponding to the element connectivities. An example of a small triangular mesh with polynomial degrees  $p = 2$  within each element is shown in Fig. 1. To be able to use machine optimized dense linear algebra routines, such as the BLAS/LAPACK libraries,<sup>13</sup> we represent  $\mathbf{A}$  in an efficient dense block format, see Ref. 8 for details. We note that our CDG scheme actually produces sparser off-diagonal blocks than other known methods,<sup>10</sup> which we take advantage of in our implementation. However, in the presentation here we assume for simplicity that all nonzero blocks are full dense matrices. The block with element indices  $1 \leq i, j \leq n$  will be denoted by  $\mathbf{A}_{ij}$ , with  $n$  the total number of elements.



**Figure 1.** An example mesh with elements of polynomial order  $p = 2$ , and the corresponding block matrix for a scalar problem.

It is clear that the performance of the iterative solvers will depend strongly on the timestep  $\Delta t$ . As  $\Delta t \rightarrow 0$ , the matrix  $\mathbf{A}$  reduces to a multiple of the mass matrix, which is inverted exactly by all preconditioners that we consider. However, as  $\Delta t \rightarrow \infty$ , we solve for a steady-state solution which in general is much harder and often not well-behaved. Physically, a small  $\Delta t$  means that the information propagation is local, while a large  $\Delta t$  means information is exchanged over large distances during the timestep. This effect, which is important when designing iterative methods, is even more important when we consider parallel algorithms since algorithms based on local information exchanges usually scale better than ones with global communication patterns.

### III. Preconditioning and Element Ordering

When solving the system  $\mathbf{A}\mathbf{u} = \mathbf{b}$  using Krylov subspace iterative methods, it is essential to use a good preconditioner. This amounts to finding an approximation  $\tilde{\mathbf{A}}$  to  $\mathbf{A}$  which allows for a relatively inexpensive computation of  $\tilde{\mathbf{A}}^{-1}\mathbf{p}$  for arbitrary vectors  $\mathbf{p}$ .

#### A. Block Jacobi, Gauss Seidel, and Incomplete LU

One of the simplest choices that performs reasonably for many problems is the block-diagonal, or the block-Jacobi, preconditioner

$$\tilde{\mathbf{A}}_{ij}^J = \begin{cases} \mathbf{A}_{ij} & \text{if } i = j, \\ \mathbf{0} & \text{if } i \neq j. \end{cases} \quad (10)$$

Clearly,  $\tilde{\mathbf{A}}^J$  is cheap to invert compared to  $\mathbf{A}$ , since all the diagonal blocks are independent. However, unlike the point-Jacobi iteration, there is a significant preprocessing cost in the factorizations of the diagonal blocks  $\tilde{\mathbf{A}}_{ij}^J$ , which is comparable to the cost of more complex factorizations.<sup>8</sup>

A minor modification of the block-diagonal preconditioner is the block Gauss Seidel preconditioner, which keeps the diagonal blocks plus all the upper triangular blocks:

$$\tilde{\mathbf{A}}_{ij}^{\text{GS}} = \begin{cases} \mathbf{A}_{ij} & \text{if } i \leq j, \\ \mathbf{0} & \text{if } i > j. \end{cases} \quad (11)$$

The preprocessing cost is the same as before, and solving  $\tilde{\mathbf{A}}^{\text{GS}}\tilde{\mathbf{u}} = \mathbf{p}$  is only a constant factor more expensive than for  $\tilde{\mathbf{A}}^J$ . The Gauss Seidel preconditioner can perform well for some simple problems, such as scalar convection problems, but in general it only gives a small factor of improvement over the block-diagonal

preconditioner. Furthermore, the sequential nature of the triangular back-solve with  $\tilde{\mathbf{A}}^{\text{GS}}$  makes the Gauss Seidel preconditioner hard to parallelize.

A more ambitious preconditioner with similar storage and computational cost is the block incomplete LU factorization  $\tilde{\mathbf{A}}^{\text{ILU}} = \tilde{\mathbf{L}}\tilde{\mathbf{U}}$  with zero fill-in. This block-ILU(0) algorithm corresponds to block-wise Gaussian elimination, where no new blocks are allowed into the matrix. With the minor assumption that any two neighbors of an element are not neighbors of each other, this factorization can be computed with the following simple algorithm:

```

function  $[\tilde{\mathbf{L}}, \tilde{\mathbf{U}}] \leftarrow \text{IncompleteLU}(\mathbf{A}, \text{mesh})$ 
 $\tilde{\mathbf{U}} = \mathbf{A}, \tilde{\mathbf{L}} = \mathbf{I}$ 
for  $j = 1, \dots, n-1$ 
    for neighbors  $i > j$  of  $j$  in  $\text{mesh}$ 
         $\tilde{\mathbf{L}}_{ij} = \tilde{\mathbf{U}}_{ij} \tilde{\mathbf{U}}_{jj}^{-1}$ 
         $\tilde{\mathbf{U}}_{ii} = \tilde{\mathbf{U}}_{ii} - \tilde{\mathbf{L}}_{ik} \tilde{\mathbf{U}}_{ki}$ 

```

We also note here that the upper-triangular blocks of  $\tilde{\mathbf{U}}$  are identical to those in  $\mathbf{A}$ , which reduces the storage requirements for the factorization. The back-solve using  $\tilde{\mathbf{L}}$  and  $\tilde{\mathbf{U}}$  has the same sequential nature as for Gauss Seidel, but the performance can be fundamentally better.<sup>8</sup>

## B. Minimum Discarded Fill Element ordering

It is clear that the Gauss Seidel and the incomplete LU factorizations will depend on the ordering of the blocks, or the elements, in the mesh. This is because of the unsymmetric pattern in which connectivities are treated. In Ref. 8, we proposed a simple heuristic algorithm for finding appropriate element orderings. Our algorithm considers the fill that would be ignored if element  $j'$  was chosen as the pivot element at step  $j$ :

$$\Delta \tilde{\mathbf{U}}_{ik}^{(j,j')} = -\tilde{\mathbf{U}}_{ij'} \tilde{\mathbf{U}}_{j'j'}^{-1} \tilde{\mathbf{U}}_{j'k}, \quad \text{for neighbors } i \geq j, k \geq j \text{ of element } j'. \quad (12)$$

With the same argument as before on the connectivities of the elements, the matrix  $\Delta \tilde{\mathbf{U}}^{(j,j')}$  corresponds to fill that would be discarded by the ILU algorithm. In order to minimize these errors, we consider a set of candidate pivots  $j' \geq j$  and pick the one that produces the smallest fill. As a measurement of the magnitude of the fill, or the corresponding weight, we take the Frobenius matrix norm of the fill matrix:

$$w^{(j,j')} = \|\Delta \tilde{\mathbf{U}}^{(j,j')}\|_{\text{F}}. \quad (13)$$

As a further simplification, we note that

$$\|\Delta \tilde{\mathbf{U}}_{ik}^{(j,j')}\|_{\text{F}} = \|-\tilde{\mathbf{U}}_{ij'} \tilde{\mathbf{U}}_{j'j'}^{-1} \tilde{\mathbf{U}}_{j'k}\|_{\text{F}} \leq \|\tilde{\mathbf{U}}_{ij'}\|_{\text{F}} \|\tilde{\mathbf{U}}_{j'j'}^{-1} \tilde{\mathbf{U}}_{j'k}\|_{\text{F}}, \quad (14)$$

which means we can estimate the weight by simply multiplying the norms of the individual matrix blocks. By pre-multiplication of the block-diagonal, we can also avoid the matrix factor  $\tilde{\mathbf{U}}_{j'j'}^{-1}$  above.

One modification that we have done to the original algorithm in Ref. 8 is the selection of candidate pivots. Instead of considering all possible pivots  $j' \geq j$ , we have found that the quality of the ordering is improved by restricting the search to elements that are neighboring already chosen pivots. This increased locality appears to be particularly beneficial when decomposing the preconditioner into the partitions used for parallel processing, which will be described in the next section. Our final minimum discarded fill algorithm then becomes:

```

function  $p = \text{OrderingMDF}(\mathbf{A}, \text{mesh})$ 
for all neighbors  $i, j$  in  $\text{mesh}$ 
     $C_{ij} = \|\mathbf{A}_{ii}^{-1} \mathbf{A}_{ij}\|_{\text{F}}$  Reduce each block to scalar
for  $k = 1, \dots, n$ 
     $w_k = \text{ComputeWeight}(k, w, C, \text{mesh})$  Compute all weights
    elements = {} List of candidates
    for  $i = 1, \dots, n$  Main loop
        if is_empty(elements)
            elements = argmin $_j w_j$  Choose smallest fill-in
             $p_i = \text{argmin}_j w_j, j \in \text{elements}$  Choose smallest candidate fill-in
             $w_{p_i} = \infty$  Do not use  $p_i$  again
            elements = elements  $\setminus p_i$  Remove  $p_i$  from list
        for neighbors  $k$  of  $p_i$  in  $\text{mesh}, w_k \neq \infty$  Update weights
             $w_k = \text{ComputeWeight}(k, w, C, \text{mesh})$ 
            elements = elements  $\cup k$ 

function  $w_k = \text{ComputeWeight}(k, w, C, \text{mesh})$ 
 $\Delta C = 0$ 
for neighbors  $i, j$  of element  $k$  in  $\text{mesh}$ ,
    if  $i \neq j, w_i \neq \infty, w_j \neq \infty$ 
         $\Delta C_{ij} = C_{ik} C_{kj}$  Discarded fill matrix
 $w_k = \|\Delta C\|_{\text{F}}$  Fill weight

```

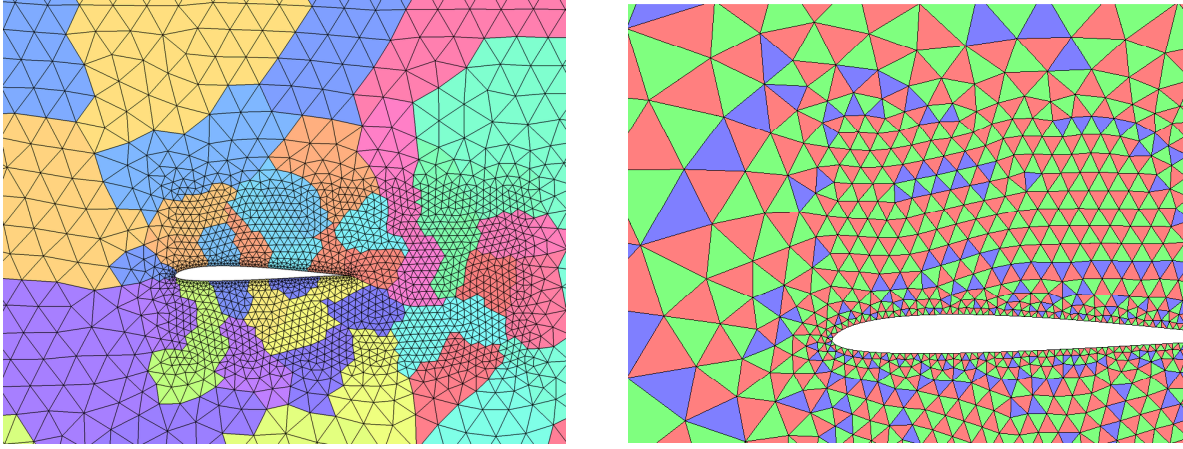
Further performance improvements include using a min-heap data structure for finding  $\text{argmin}_j w_j$  in the main loop, and running the algorithm only once as a preprocessing step with a low polynomial order. To obtain good orderings for the Gauss Seidel method, a minor modification is required in the weight calculations.<sup>8</sup>

We note that for a pure upwinded scalar convective problem, the MDF ordering is optimal since at each step it picks an element that either does not affect any other elements (downwind) or does not depend on any other elements (upwind), resulting in a perfect factorization. But the algorithm works well for other problems too, including multivariate and viscous problems, since it tries to minimize the error between the exact and the approximate LU factorizations. It also takes into account the effect of the discretization (e.g. highly anisotropic elements) on the ordering. These aspects are harder to account for with methods that are based on physical observations, such as line-based orderings.<sup>14–16</sup>

## IV. Parallelization Strategies

The parallelization of the discretization and the assembly parts of our discontinuous Galerkin solver is overall fairly straight-forward. The domain is naturally partitioned by the elements to achieve load balancing and low communication volumes, see Fig. 2 (left). Because of the element-wise compact stencil of the CDG scheme, only one additional layer of elements has to be maintained for each process in addition to the elements in the partition. The elements at the domain boundary are processed first, then the computed data is sent to neighboring processes using non-interrupting communications, and while the data is sent the interior elements are processed. This typically leads to algorithms where the communication costs are negligible for evaluations of the residual  $\mathbf{r}(\mathbf{u})$ , and therefore also for explicit time integration methods, as well as for the evaluation of the Jacobian matrix  $\mathbf{K} = \partial \mathbf{r} / \partial \mathbf{u}$ .

Note that the reason medium- and high-order discontinuous Galerkin codes scale well on parallel computers is not necessarily the element-wise compact stencil. The total amount of data communicated is comparable with similar high-order finite difference or finite volume methods. Instead, it is because of the high *cost* of assembly, involving integration of high-degree polynomials, that the communication costs become



**Figure 2. Partitioning (left) and coloring (right) of the mesh elements.**

less significant.

In the iterative implicit solvers, the matrix-vector products also scale well in terms of communication, by overlapping with the computation of the interior elements. However, a problem here is the fact that these operations are highly memory intensive, with only about 2 floating point operations per matrix entry. This tends to give poor scaling on multicore processors, which are becoming the standard in cluster computers. Another issue is the parallelization of the preconditioner, where in particular the Gauss Seidel and the incomplete LU preconditioners have a highly serial structure that is hard to parallelize.

### A. Domain Decomposition

Our preferred approach for parallelization of the ILU factorization is to apply it according to the element orderings determined by the MDF algorithm, but ignoring any contributions between elements in different partitions. In standard domain decomposition terminology, this essentially amounts to a non-overlapping Schwarz algorithm with incomplete solutions.<sup>17</sup> It is clear that this approach will scale well in parallel, since each process will compute a partition-wise ILU factorization independent of all other processes.

To minimize the error introduced by separating the ILU factorizations, we use the ideas from the MDF algorithm to obtain information about the weights of the connectivities between the elements. By computing a weighted partitioning using the weight

$$C_{ij} = \|\mathbf{A}_{ii}^{-1} \mathbf{A}_{ij}\|_F \quad (15)$$

between element  $i$  and  $j$ , we obtain partitions that are less dependent on each other, and reduce the error from the decomposition. The drawback is that the total communication volume might be increased, but if desired, a trade-off between these two effects can be obtained by adding a constant  $C_0$  to the weights. In practice, since the METIS software<sup>18</sup> used for the partitioning requires integer weights, we scale and round the  $C_{ij}$  values to integers between 1 and 100.

It is clear that this method reduces to the block-Jacobi method as the number of partitions approaches the number of elements. However, in any practical setting, each partition will have at least 100's of elements, in which the difference between partition-wise block-ILU and block-Jacobi is quite significant.

### B. Graph Coloring

An alternative approach, which is standard for parallelization of ILU factorizations,<sup>19</sup> is to *color* the graph to find large independent sets of elements. In a DG setting, this is naturally done on the element level, see Fig. 2 (right). All elements of equal color can be updated in parallel, independently of each other. In this setting, we note an advantage of DG over standard continuous Galerkin FEM methods, in that all the degrees in the DG element graph are low (maximum  $D + 1$  for simplex elements in  $D$  dimensions). This

makes is straight-forward to find colorings with few colors – a simple greedy algorithm typically gives the optimal  $D + 1$  number of colors:

```

function  $c = \text{GraphColoring}(\text{mesh})$ 
 $c_i = 0, \quad i = 1, \dots, n$ 
for  $i = 1, \dots, n$ 
     $c_{\text{neighbors}} = \{c_j \mid j \text{ neighbor of } i \text{ in } \text{mesh}\}$ 
     $c_i = \min\{j \mid j \geq 1, j \notin c_{\text{neighbors}}\}$ 

```

However, as we will see in Section V, the performance of the ILU preconditioner is often poor with these orderings, in particular for convection-dominated problems where the element ordering has a large influence. This is easy to understand, at least conceptually, from the mechanisms of the ILU algorithm and the MDF ordering. The ordering from the colored graph will choose pivots whose neighbors of higher color have not yet been processed. This will typically increase the magnitude of the fill, which is discarded by the ILU algorithm, since the fill weights  $w_k$  will include contributions from more neighboring elements. In contrast, the goal of the MDF algorithm is to minimize these weights, which usually results in picking elements whose neighbors already have been processed. In this sense, the coloring does the complete opposite of what we expect from the ordering. Instead of finding elements whose dependencies are sequentially aligned, the coloring will entirely avoid sequences of neighboring elements.

## V. Results

In this section, we apply the methods on representative test problems and show convergence and timing results. We focus on the solution of the linear system of equations (9) using iterative Krylov subspace methods, and do not consider issues regarding the convergence of the Newton method. In all examples, we iterate until we obtain a relative accuracy of  $10^{-4}$  in the solution.

### A. Convergence Properties

First, we study the convergence properties of the proposed schemes, and in particular how the parallelization of the algorithms affect the number of iterations required for convergence. The test problem is the planar flow around an SD7003 foil, at Mach number 0.1, Reynolds number 8000, and with an angle of attack of 3 degrees (Fig. 3). The mesh is a structured C-mesh, with 3936 triangular elements and anisotropy ratios up to 100:1 at the boundary. The solution is approximated by polynomials of degree  $p = 3$  within each element. The CFL-limit of an explicit Runge-Kutta fourth order integrator corresponds to a timestep less than  $\Delta t < 10^{-6}$ , and an implicit solution procedure is likely the only practical alternative. For the linear solver, we use a restarted GMRES solver<sup>19,20</sup> with a restart value of 20.

The convergence results are summarized in the graphs in Fig. 4. We plot the number of iterations versus the timestep  $\Delta t$  for a number of preconditioners: **Jacobi** is the Block-Jacobi method (10), and **Colored ILU** is the Block-ILU method  $\tilde{A}^{\text{ILU}} = \tilde{L}\tilde{U}$ , with elements ordered by the colors produced by the coloring algorithm. **Partition ILU** is the block-ILU method applied partition-wise, with a standard unweighted partitioning. The elements are ordered according to the MDF ordering within each partition. The **Weighted Partition ILU** is like the partition ILU, but with partitioning based on the weights  $C_{ij}$  in (15). Finally, the **Serial ILU** preconditioner is the regular serial block-ILU with MDF ordering from Ref. 8.

We can draw the following conclusions from the convergence plots:

- The block-Jacobi preconditioner shows very poor convergence except when the timestep is small.
- The coloring of the elements in the block-ILU preconditioner appears to destroy the excellent convergence properties obtained using the MDF ordering. For medium and large timestep, the scheme does not perform much better than block-Jacobi.
- The partitioning of the block-ILU preconditioner increases the number of iterations by approximately a constant factor. This effect is somewhat reduced by using the weighted partitioning.

Based on these results, we will not consider the coloring strategy (we have not even implemented it in parallel), and we will only use the weighted partitioning for the ILU preconditioner.

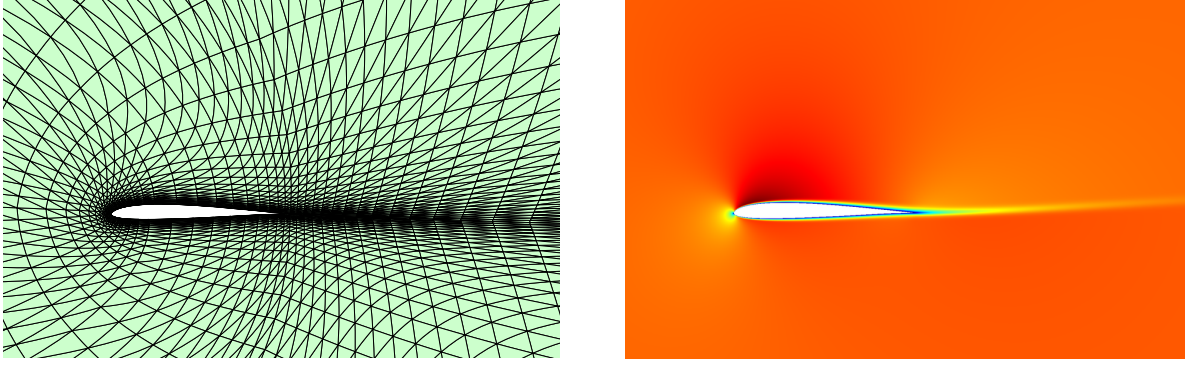


Figure 3. The SD7003 test problem, mesh (left) and Mach number plot (right).

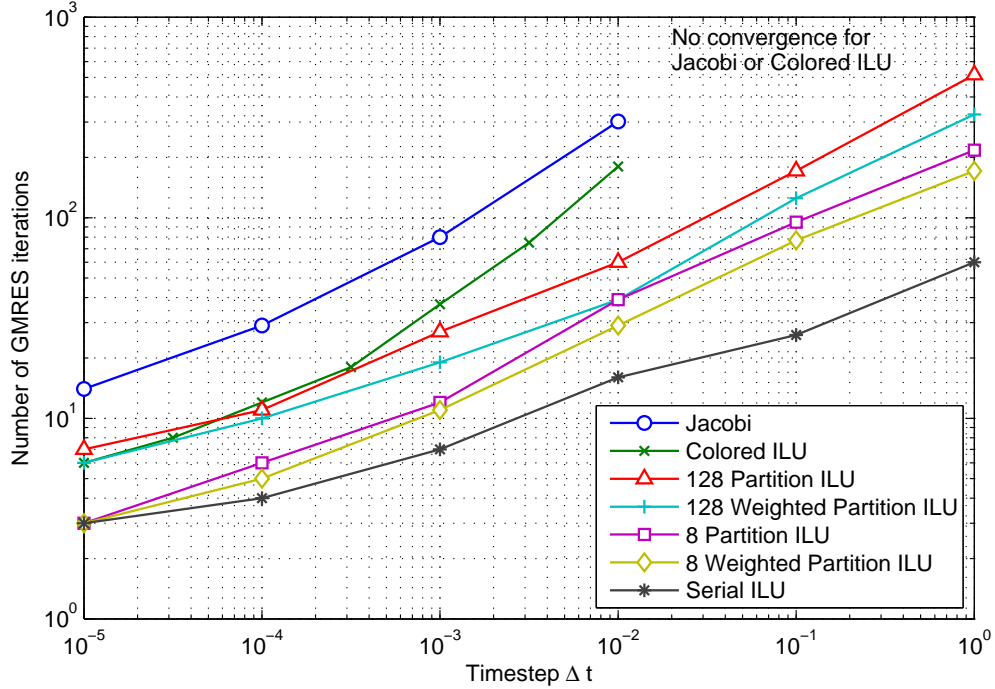


Figure 4. Convergence results for the SD7003 test problem for various preconditioners and partitioners.

## B. Execution times and Speedup

Next, we study the actual execution times of our parallel implementations. Our test computer is a Linux cluster with two Xeon Intel quad-core 64-bit processors, or 8 cores, in each compute node, running at a clock speed of 2.66GHz. The high performance, low latency interconnect provides a bandwidth of 20Gb/s. In our parallel implementation, we use the Conjugate Gradient Squared (CGS) algorithm as the iterative Krylov subspace solver.

First, we show the performance of a simple explicit RK4 algorithm, which is a measurement of how well a regular residual evaluation is parallelized. The test problem is the unsteady flow around an elliptic wing, at Mach number 0.3, Reynolds number 2000, and 30 degrees angle of attack (Fig. 5). The mesh contains about 190,000 tetrahedral elements with polynomial degrees  $p = 3$ , giving a total of about 19 million degrees of freedom.

The speedup (serial time divided by parallel time) is shown in Fig. 6. We note that already at 8



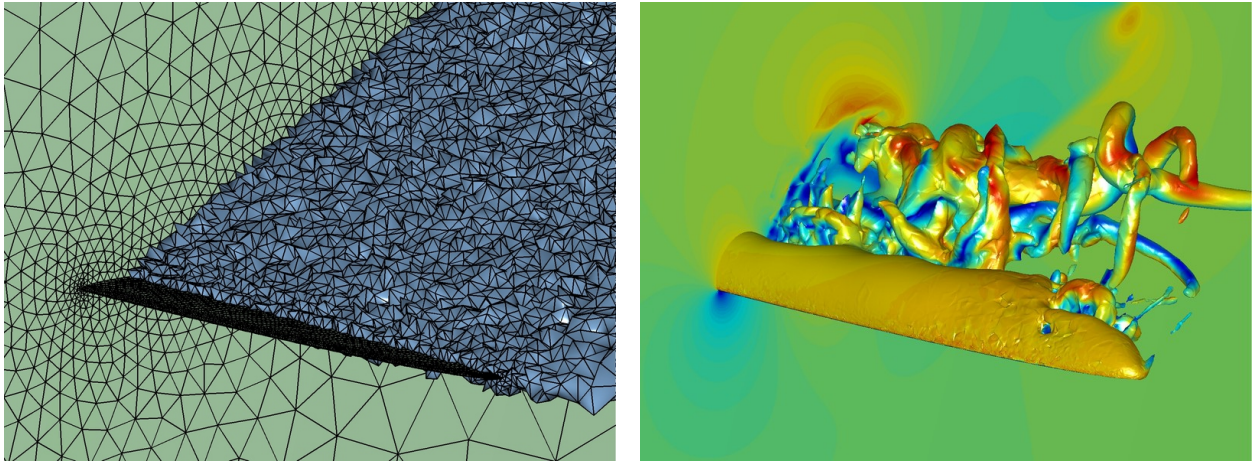
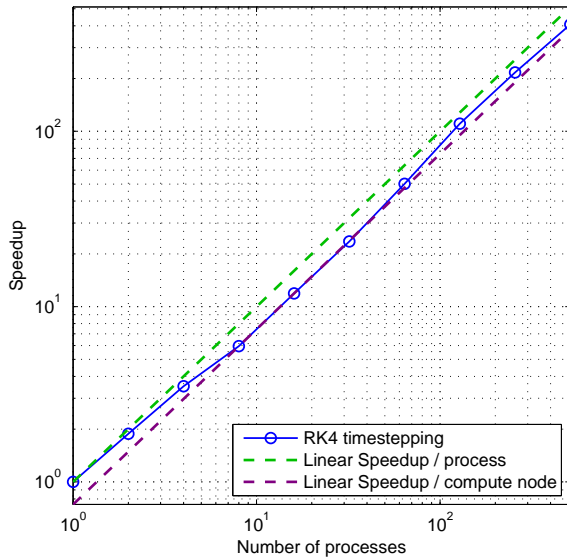


Figure 5. The elliptic wing test problem. The mesh consists of 190,000 tetrahedra of polynomial degrees  $p = 3$  (left), giving a total of about 19 million degrees of freedom. The solution is computed using a 3rd order accurate implicit Runge-Kutta scheme, and shown at time  $t = 10.0$  as Mach number plotted on isosurfaces of the pressure (right).



Processes	Speedup	Efficiency
1	1.0	1.00%
2	1.8	0.94%
4	3.5	0.88%
8	5.9	0.74%
16	11.8	0.74%
32	23.5	0.74%
64	50.1	0.78%
128	110.6	0.86%
256	216.6	0.85%
512	406.7	0.79%

Figure 6. Speedup for RK4 explicit time integration of the elliptic wing problem. The deviation from linear speedup per process is mainly due to the limited memory bandwidth for the 8 processor cores in each compute node. The speedup for increasing number of nodes is close to ideal, due to the overlap between communication and computation.

processes, there is about a 25% decrease in efficiency. This is not due to the “classical parallelization problem” of communication cost, since all 8 processes run on a single shared memory node. Instead, this demonstrates the limitations with the limited memory bandwidth of multicore computers (which will be even more pronounced for matrix-vector products). When the number of nodes is increased, the speedup per compute node is essentially linear, showing that communication time is not a problem if implemented correctly with overlapping computations.

Next, we consider the solution of the linear system of equations (9) for the SD7003 test problem in Fig. 3. The speedup is shown in Fig. 7 for the assembly, linear solution, and total compute times separately. In the left plot, we see that the assembly process scales almost perfectly, which is expected because of two reasons: 1) The communication costs are negligible compared to the assembly of the Jacobian matrix, and 2) The operation has a high ratio of computation / memory access, which gives good scaling within each multicore

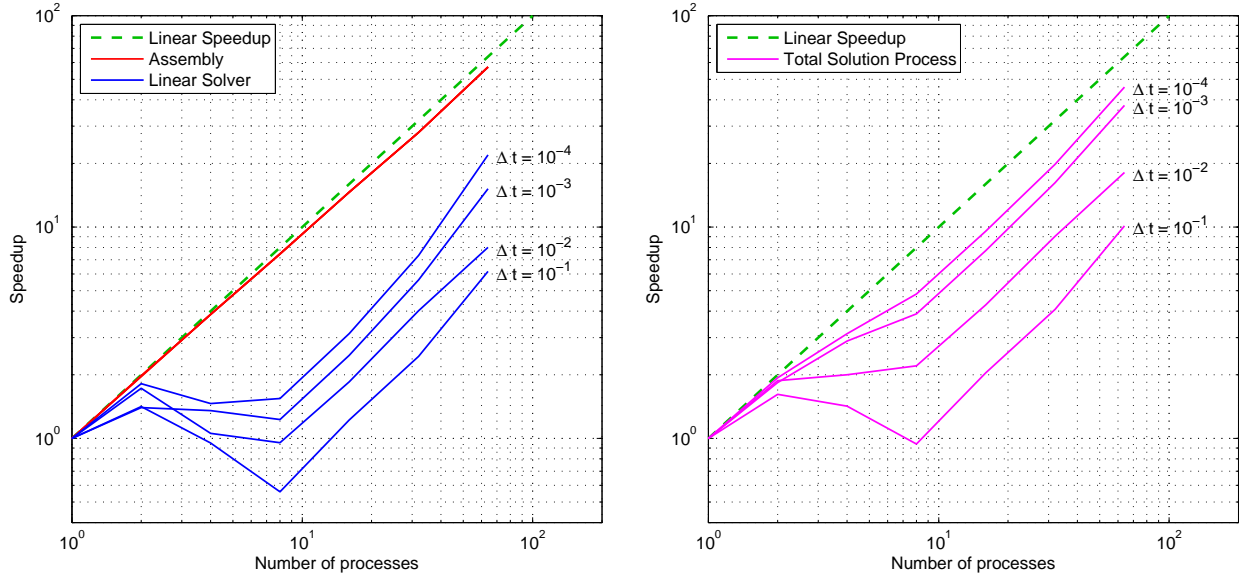


Figure 7. The speedup for solving the linear systems in the planar SD7003 test problem. We note that the linear solution process is highly limited by the memory bandwidth within each compute node. Combined with the increase in Krylov iterations due to partitioning, the speedup for 8 processes is poor. However, for increasing number of compute nodes the algorithm scales well. As a result, the total solution time and speedup depend strongly on the timestep.

compute node.

In contrast, the linear solution times scale poorly up to 8 processes, due to two reasons: 1) The number of iterations increase due to the partitioning of the ILU preconditioner, and 2) This process is dominated by the time for matrix-vector products and triangular back-solves, which are memory intensive operations that are limited by the memory bandwidth within each compute node. However, when increasing the number of nodes the speedup per compute node is close to linear, showing again that communication is not a major problem. This also demonstrates that partitioning of the ILU preconditioner is a feasible approach for obtaining scalable solvers.

The timings of the assembly process and the linear solution process combine to give the total solution time (right plot). As expected, the assembly process dominates for smaller timesteps, giving good speedup, but the linear solution time dominates for larger timesteps, giving poor speedup.

Comparing the partitioned ILU preconditioner with block-Jacobi, see Fig. 8, we see that even for modest timesteps block-ILU outperforms block-Jacobi. This is due to the fact that Jacobi is a poor preconditioner, which was observed in Ref. 8, and this effect is not compensated for by its good scalability.

As a final example, we show the results from solving the linear systems of the elliptic wing problem in Fig. 5. This is large problem, and the storage requirements for the Jacobian matrix is about 36GB, plus a little more than half of this for the ILU factorization. Therefore, at least 8 compute nodes are needed, so the smallest number of processes we use is 64. The speedup (with respect to the 64 process run) and timings for 64, 128, 256, and 512 processes are shown in Fig. 9.

We note that the speedup is essentially perfect, except possibly for the larger timestep. Compared to the Jacobi preconditioner, ILU is more competitive for larger timesteps, but for smaller timesteps the difference is small since the solution time is dominated by the assembly process. The reason that the differences between the schemes are smaller for this example than for the SD7003 problem is likely the fact that this problem is better behaved, with lower Reynolds number and an isotropic mesh, which gives the Jacobi preconditioner reasonably good convergence.

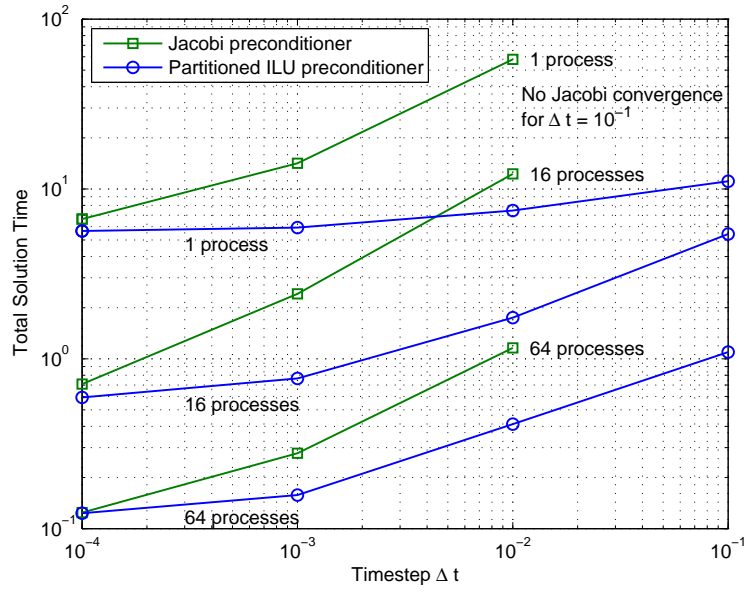


Figure 8. Total compute times for the SD7003 problem, using the partitioned ILU and the block-Jacobi preconditioners. For small timesteps the differences are small, but as  $\Delta t$  increases the Jacobi performance deteriorates which results in very poor performance.

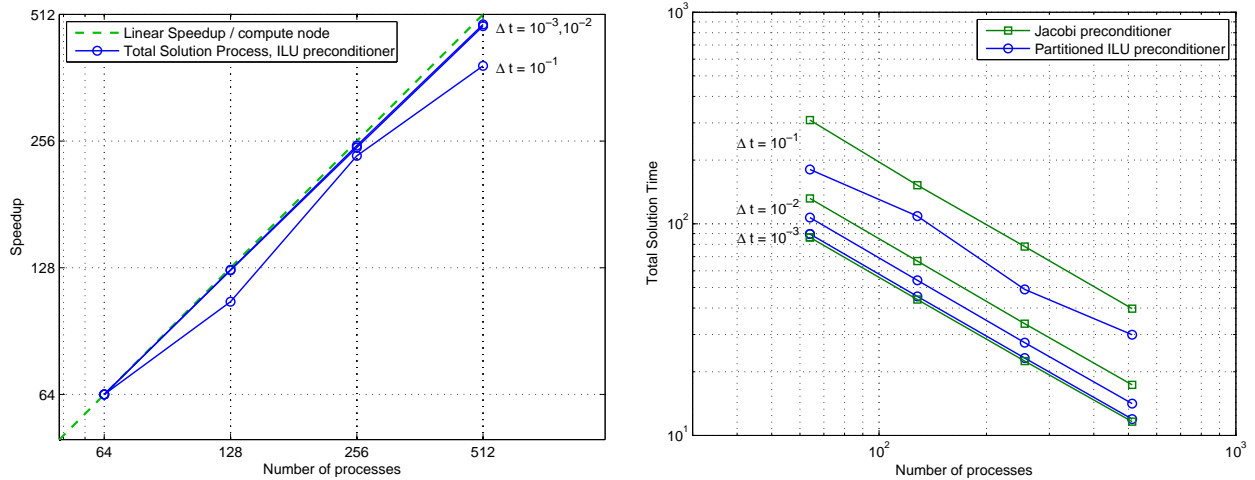


Figure 9. The speedup and timing for the elliptic wing problem. This problem is too large to run in serial, but almost perfect speedup per compute node is obtained (left). For the timings (right), we see that for small timesteps the solution process is dominated by the assembly process. This makes the difference between the Jacobi and the ILU preconditioner small. However, for larger timesteps the ILU preconditioner is significantly more effective.

## VI. Conclusions

We have presented our work on parallel solvers for implicit discontinuous Galerkin methods. Although ILU-based preconditioners do not scale perfectly on parallel computers, they can still outperform simpler alternatives such as block-Jacobi due to their fundamentally better convergence properties. A simple domain decomposition approach, together with optimal MDF element orderings within each domain, gives good scaling even for as many as 512 processes.

We also note that matrix-based iterative solvers scale somewhat poorly on multicore compute nodes, due to the limited memory bandwidth. This motivates to some extent the use of more complex preconditioners that would reduce the number of iterations further. In particular, the preconditioner we proposed in Ref. 8, combining ILU with a coarse grid correction, provides excellent convergence for a wide range of flow problems. Future work includes the study of how to parallelize this preconditioner.

## VII. Acknowledgements

I would like to acknowledge my long standing collaboration with Jaime Peraire from the Massachusetts Institute of Technology, and our multiple discussions regarding this work. I am also grateful to the Scientific Cluster Support group at the Lawrence Berkeley National Laboratory, for providing parallel computing resources and user support.

## References

- <sup>1</sup>Cockburn, B. and Shu, C.-W., “Runge-Kutta discontinuous Galerkin methods for convection-dominated problems,” *J. Sci. Comput.*, Vol. 16, No. 3, 2001, pp. 173–261.
- <sup>2</sup>Nguyen, N. C., Persson, P.-O., and Peraire, J., “RANS Solutions Using High Order Discontinuous Galerkin Methods,” *45th AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, 2007*, AIAA-2007-914.
- <sup>3</sup>Bassi, F. and Rebay, S., “A high-order accurate discontinuous finite element method for the numerical solution of the compressible Navier-Stokes equations,” *J. Comput. Phys.*, Vol. 131, No. 2, 1997, pp. 267–279.
- <sup>4</sup>Cockburn, B. and Shu, C.-W., “The Runge-Kutta discontinuous Galerkin method for conservation laws. V. Multidimensional systems,” *J. Comput. Phys.*, Vol. 141, No. 2, 1998, pp. 199–224.
- <sup>5</sup>Hartmann, R. and Houston, P., “An optimal order interior penalty discontinuous Galerkin discretization of the compressible Navier-Stokes equations,” *J. Comput. Phys.*, Vol. 227, No. 22, 2008, pp. 9670–9685.
- <sup>6</sup>B. Landmann, M. Kessler, S. W. and Krämer, E., “A Parallel Discontinuous Galerkin Code for the Navier-Stokes Equations,” *44th AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, 2006*, AIAA-2006-111.
- <sup>7</sup>Nastase, C. and Mavriplis, D., “A Parallel hp-Multigrid Solver for Three-Dimensional Discontinuous Galerkin Discretizations of the Euler Equations,” *45th AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, 2007*, AIAA-2007-512.
- <sup>8</sup>Persson, P.-O. and Peraire, J., “Newton-GMRES preconditioning for discontinuous Galerkin discretizations of the Navier-Stokes equations,” *SIAM J. Sci. Comput.*, Vol. 30, No. 6, 2008, pp. 2709–2733.
- <sup>9</sup>Roe, P. L., “Approximate Riemann solvers, parameter vectors, and difference schemes,” *J. Comput. Phys.*, Vol. 43, No. 2, 1981, pp. 357–372.
- <sup>10</sup>Peraire, J. and Persson, P.-O., “The compact discontinuous Galerkin (CDG) method for elliptic problems,” *SIAM J. Sci. Comput.*, Vol. 30, No. 4, 2008, pp. 1806–1824.
- <sup>11</sup>Shampine, L. F. and Gear, C. W., “A user’s view of solving stiff ordinary differential equations,” *SIAM Rev.*, Vol. 21, No. 1, 1979, pp. 1–17.
- <sup>12</sup>Alexander, R., “Diagonally implicit Runge-Kutta methods for stiff o.d.e.’s,” *SIAM J. Numer. Anal.*, Vol. 14, No. 6, 1977, pp. 1006–1021.
- <sup>13</sup>Anderson, E. et al., *LAPACK Users’ Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd ed., 1999.
- <sup>14</sup>Nastase, C. R. and Mavriplis, D. J., “High-order discontinuous Galerkin methods using an hp-multigrid approach,” *J. Comput. Phys.*, Vol. 213, No. 1, 2006, pp. 330–357.
- <sup>15</sup>Fidkowski, K., Oliver, T., Lu, J., and Darmofal, D., “p-Multigrid solution of high-order discontinuous Galerkin discretizations of the compressible Navier-Stokes equations,” *J. Comput. Phys.*, Vol. 207, No. 1, 2005, pp. 92–113.
- <sup>16</sup>Kanschä, G., “Robust smoothers for high-order discontinuous Galerkin discretizations of advection-diffusion problems,” *J. Comput. Appl. Math.*, Vol. 218, No. 1, 2008, pp. 53–60.
- <sup>17</sup>Toselli, A. and Widlund, O., *Domain Decomposition Methods - Algorithms and Theory*, Vol. 34 of *Springer Series in Computational Mathematics*, Springer, 2004.
- <sup>18</sup>Karypis, G. and Kumar, V., “METIS Serial Graph Partitioning and Fill-reducing Matrix Ordering,” <http://glaros.dtc.umn.edu/gkhome/metis/metis/overview>.
- <sup>19</sup>Saad, Y., *Iterative methods for sparse linear systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2nd ed., 2003.
- <sup>20</sup>Saad, Y. and Schultz, M. H., “GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems,” *SIAM J. Sci. Statist. Comput.*, Vol. 7, No. 3, 1986, pp. 856–869.